

Práctica Entregable: Diseño de Aplicaciones en la Nube

API: Diseño de aplicaciones básicas en la nube



Daniel Moreno López

ÍNDICE

INTRODUCCIÓN.....	3
OBJETIVOS.....	4
FUNCIONALIDAD.....	5
CICLO DE VIDA.....	5
INFRAESTRUCTURA DESPLEGADA.....	5
AUTOMATIZACIÓN.....	6
PRUEBAS.....	6
TECNOLOGÍAS USADAS.....	7
INFRAESTRUCTURA COMO CÓDIGO(IaC).....	7
SERVICIOS DE CÓMPUTO Y CONTENEDORES.....	7
AWS FARGATE(OPCIÓN A).....	7
AWS LAMBDA(OPCIÓN B).....	7
DOCKER.....	8
ECR.....	8
RED ACCESO Y SEGURIDAD.....	8
API GATEWAY.....	8
NETWORK LOAD BALANCER(NLB).....	8
VPC LINK.....	9
BASE DE DATOS Y ALMACENAMIENTO.....	9
DYNAMODB.....	9
S3(Simple Storage Service).....	9
DESARROLLO Y AUTOMATIZACIÓN.....	9
DISEÑO E IMPLEMENTACIÓN.....	10
ARQUITECTURA COMÚN.....	10
Base de Datos(DynamoDB):.....	10
API GATEWAY:.....	11
OPCIÓN A: API MONOLÍTICA CON AWS FARGATE/CONTENEDORES.....	11
OPCIÓN B: API DE MICROSERVICIOS CON AWS LAMBDA.....	13
PUESTA EN MARCHA Y VERIFICACIÓN DE DESPLIEGUE.....	14
PRERREQUISITOS:.....	14
PYTHON 3 Y PIP.....	14
BOTO3.....	14
DOCKER DESKTOP(SOLO PARA LA OPCIÓN A).....	14
AWS CLI Y CREDENCIALES CONFIGURADAS.....	15
DESPLIEGUE AUTOMATIZADO.....	17
DESPLIEGUE ECS FARGATE.....	17
DESPLIEGUE AWS LAMBDA.....	23
VERIFICACIÓN EN LA CONSOLA DE AWS.....	26
STACKS DE CLOUDFORMATION.....	26

SALIDAS DE LOS STACKS.....	27
PRUEBAS Y VERIFICACIÓN.....	28
VERIFICACIÓN ENDPOINTS CON POSTMAN.....	28
PRUEBA CREACION(POST /notes).....	29
PRUEBA DE LECTURA (GET /notes).....	29
PRUEBA DE ACTUALIZACIÓN(PUT /notes/id).....	30
PRUEBA DE BORRADO(DELETE /notes/id).....	31
VERIFICACIÓN FUNCIONAL CON FRONTEND.....	32
CONFIGURACIÓN DEL CLIENTE WEB.....	32
PRUEBA FUNCIONALIDAD CRUD.....	33
ANÁLISIS DE COSTOS(PRICING).....	36
ESCENARIO DE ESTIMACIÓN.....	36
DESGLOSE DE COSTOS.....	37
COSTOS COMUNES.....	37
COSTOS ESPECÍFICOS: OPCIÓN A(FARGATE).....	37
COSTOS ESPECÍFICOS: OPCIÓN B(LAMBDA).....	38
COMPARATIVA FINAL DE COSTOS.....	38
COMPARATIVA DE ARQUITECTURAS.....	39
CONCLUSIONES FINALES.....	40
USO DE LA IA.....	41
BIBLIOGRAFÍA.....	42

INTRODUCCIÓN

En este informe se comentará el diseño, implementación y análisis comparativo de una aplicación web serverless en la nube de Amazon Web Services (AWS). El proyecto se centra en la creación de una API RESTful para un servicio de "Notas", implementando dos arquitecturas distintas pero que comparten una base de datos y un punto de entrada de API común.

También se analizarán: los contenedores bajo demanda (AWS Fargate) y las funciones como servicio (AWS Lambda). A través de la automatización completa del despliegue mediante Infraestructura como Código (IaC) con AWS CloudFormation.

OBJETIVOS

El objetivo principal es "diseñar y desplegar una aplicación robusta y escalable utilizando servicios fundamentales de AWS". Para alcanzar esta meta, se han establecido los siguientes objetivos específicos:

- Desplegar y Comparar Dos Arquitecturas:
 - **Opción A (Contenedores):** Implementar un *backend* basado en contenedores Docker, desplegado en un servicio de computación *serverless* como AWS Fargate incluyendo un Network Load Balancer (NLB) y un VPC Link para la integración privada con API Gateway.
 - **Opción B (Funciones):** Implementar un *backend* de microservicios basado en AWS Lambda, adoptando el modelo de "arquitectura desacoplada". Cada operación CRUD se gestiona mediante una función Lambda independiente
- Automatizar la Gestión de la Infraestructura (IaC): Definir toda la infraestructura de AWS (bases de datos, clústeres, APIs, funciones) de forma declarativa utilizando plantillas de AWS CloudFormation
- Centralizar y Asegurar el Acceso: Utilizar Amazon API Gateway como punto de entrada único y obligatorio para ambas arquitecturas.

FUNCIONALIDAD

CICLO DE VIDA

El proyecto abarca todo el ciclo de vida de la aplicación, desde la definición de la infraestructura hasta la prueba funcional del *frontend*.

Funcionalidad de la API: La API de "Notas" implementa los siguientes cinco endpoints, tal como se especifica en los requisitos de la práctica:

- POST /notes: Crear una nueva nota.
- GET /notes: Obtener todos los elementos (Leer Todo).
- GET /notes/{id}: Obtener una nota por su ID (Leer).
- PUT /notes/{id}: Actualizar una nota existente (Actualizar).
- DELETE /notes/{id}: Eliminar una nota (Eliminar)

INFRAESTRUCTURA DESPLEGADA

El proyecto despliega los siguientes componentes en AWS:

- Común: DynamoDB, para la base de datos, un Amazon API Gateway con *stages* dinámicos y API Key), y un repositorio Amazon ECR (para la imagen Docker).
- Opción A (Fargate): Un clúster de ECS, una definición de tarea, un servicio de Fargate, un Network Load Balancer (NLB) y un VPC Link.
- Opción B (Lambda): Cinco funciones AWS Lambda, un *bucket* S3 (para el código fuente de las Lambdas) y los permisos de IAM correspondientes.

AUTOMATIZACIÓN

El proyecto incluye un conjunto de scripts de Python para automatizar tareas clave:

- `deploy-dynamodb.py`, `deploy-ecr.py`, `deploy-ecs.py`, `deploy-lambda.py`: Despliegue de los stacks de CloudFormation.
- `package-lambdas.py`: Script de build para empaquetar las funciones Lambda con su código compartido (`shared/`) y dependencias (`requirements.txt`).
- `push-image.py`: Script para construir y subir la imagen Docker de la Opción A a ECR.

PRUEBAS

El alcance de la validación incluye pruebas funcionales de la API mediante una colección de Postman y el uso de una interfaz web rudimentaria

TECNOLOGÍAS USADAS

INFRAESTRUCTURA COMO CÓDIGO(IaC)

CloudFormation es el pilar fundamental del proyecto. Toda la infraestructura de AWS, desde la base de datos hasta las funciones Lambda y los balanceadores de carga, se ha definido de forma declarativa en plantillas YAML (01-dynamodb.yml, 03-ecs-option-a.yml, 04-lambda-option-b.yml). Esto garantiza un despliegue y una eliminación consistentes, reproducibles y automatizados, eliminando la configuración manual.

SERVICIOS DE CÓMPUTO Y CONTENEDORES

AWS FARGATE(OPCIÓN A)

Es el motor de cómputo serverless para contenedores utilizado en la primera arquitectura. Se define mediante una **AWS::ECS::TaskDefinition** (que especifica la imagen Docker y los recursos) y un **AWS::ECS::Service** (que mantiene la tarea en ejecución). Fargate elimina la necesidad de gestionar los servidores (instancias EC2) subyacentes del clúster de ECS.

AWS LAMBDA(OPCIÓN B)

Es el servicio de cómputo serverless basado en Funciones como Servicio (FaaS). En esta arquitectura, cada endpoint de la API (Notes, get_note, etc.) se mapea a una **AWS::Lambda::Function** independiente. Este es el enfoque de microservicios puros, donde el código solo se ejecuta en respuesta a un evento (la llamada de API Gateway) y el costo es cero en caso de no haber tráfico.

DOCKER

Utilizado en la Opción A para empaquetar la aplicación web Python (Flask) en una imagen de contenedor. El archivo DockerFile define el entorno de ejecución basado en python:3.11-slim.

ECR

Servicio de AWS para almacenar, gestionar y desplegar las imágenes Docker, definido en el stack 02-ecr.yml. El script push-image.py automatiza la subida de la imagen a este repositorio.

RED ACCESO Y SEGURIDAD

API GATEWAY

Es el servicio central y obligatorio que actúa como la "puerta de entrada" unificada para ambas arquitecturas. EL API GATEWAY se encarga de:

- Exponer los endpoints HTTP
- Proporcionar seguridad mediante el uso de API Keys
- Enrutar el tráfico al backend correspondiente: mediante VPC Link o integración nativa

NETWORK LOAD BALANCER(NLB)

Utilizado exclusivamente en la Opción A (Fargate). Dado que el servicio de Fargate se ejecuta en una VPC privada, se requiere un NLB internal para exponer el puerto del contenedor (8080) a otros servicios dentro de la VPC.

VPC LINK

Es el "túnel" privado que conecta el API Gateway con el Network Load Balancer.

BASE DE DATOS Y ALMACENAMIENTO

DYNAMODB

Es la base de datos NoSQL serverless elegida. La tabla Notes se define en 01-dynamodb.yml con un modo de facturación PAY_PER_REQUEST, lo que la convierte en una solución rentable y escalable.

S3(Simple Storage Service)

El script deploy-lambda.py sube los paquetes de código .zip de las funciones a un bucket S3. La plantilla de CloudFormation de Lambda crea este bucket y las funciones cargan su código desde él.

DESARROLLO Y AUTOMATIZACIÓN

Para la parte de desarrollo se ha elegido el lenguaje de programación de Python para tanto escribir la aplicación web de la opción A, la lógica de negocio de las 5 funciones lambda y todos los scripts de automatización y despliegue. Cabe destacar que también se usa tanto Boto3(SDK oficial de AWS) como Pydantic para la interacción con la API de AWS y obtención de código compartido de la opción B para la validación de los datos que entran y salen de la API.

El stack se validó usando Postman para pruebas de API directas y el frontend notes-frontend.html para la validación funcional más cercana a la experiencia de usuario.

DISEÑO E IMPLEMENTACIÓN

El diseño del proyecto se ha dividido en una infraestructura común que sirve de base para ambas arquitecturas (base de datos y frontend). La Opción A es una arquitectura monolítica basada en contenedores serverless. La Opción B, una arquitectura de microservicios basada en funciones serverless.

Se ha decidido usar AWS CloudFormation manejando los scripts de Python mencionados anteriormente para evitar la creación de recursos de manera manual a través de la consola de AWS, lo cual es un proceso propenso a errores y no reproducible con la misma facilidad que la manera propuesta, con el fin de definir cada componente en plantillas YAML. Los scripts actúan como un motor de despliegue que lee dichas plantillas y gestiona el ciclo de vida de los stacks de AWS.

ARQUITECTURA COMÚN

Ambas opciones de backend (Fargate y Lambda) fueron diseñadas para ser intercambiables sin que el cliente note la diferencia. Para lograr esto, comparten dos componentes clave:

Base de Datos(DynamoDB):

En cuanto al diseño se definió una única tabla NoSQL llamada **Notes** en Amazon DynamoDB. La plantilla 01-dynamodb.yml define su esquema con una clave de partición simple.

Se eligió DynamoDB por encima de RDS por la facilidad encontrada en cuanto a su implementación, su naturaleza serverless se alinea con los dos modelos de cómputo y por el modo de facturación `BillingMode:PAY_PER_REQUEST` el cual su costo es 0 si no hay tráfico, escalando automáticamente para manejar picos de demanda

API GATEWAY:

Se utiliza como único punto de entrada, obligatoria para ambas arquitecturas. Es responsable de definir los endpoints y métodos HTTP. Esta decisión es crucial para el desacoplamiento y la seguridad por los siguientes motivos:

- Seguridad: API Gateway gestiona la autenticación de forma centralizada mediante `ApiKeyRequired: true` en las plantillas.
- Desacoplamiento: El cliente solo conoce la URL de API Gateway y no sabe si detrás hay un contenedor Fargate o una función Lambda. Podemos cambiar la `ApiUrl` en el frontend y la aplicación sigue funcionando, probando que el backend es intercambiable.
- Stages Dinámicos: Para evitar conflictos durante el despliegue, se implementó un sistema de stages dinámicos. El script `deploy-ecs.py` genera un nombre de stage único basado en la fecha y hora y lo pasa como parámetro (`StageName`) a la plantilla de CloudFormation.

OPCIÓN A: API MONOLÍTICA CON AWS FARGATE/CONTENEDORES

Esta arquitectura implementa la API como una única aplicación web (un "monolito") que se ejecuta 24/7 dentro de un contenedor *serverless*.

Se trata de una aplicación Python que utiliza el micro-framework Flask (`app-ecs/main.py`). Esta única aplicación define todas las rutas y la lógica de negocio para conectarse a DynamoDB. La aplicación se empaqueta en una imagen de Docker usando el `DockerFile` y se sube a un repositorio Amazon ECR mediante el script `push-image.py`.

El stack `03-ecs-option-a.yml` define una arquitectura de red compleja para conectar de forma segura el API Gateway al contenedor.

El flujo sería el siguiente:

1. El usuario llama a API Gateway.
2. API Gateway pasa la petición a un `AWS::ApiGateway::VpcLink`.
3. El `VpcLink` la envía a un `AWS::ElasticLoadBalancingV2::LoadBalancer` (NLB).
4. El NLB enruta el tráfico a una tarea de AWS Fargate (`AWS::ECS::Service`).
5. La tarea de Fargate ejecuta el contenedor Docker con la aplicación Flask.
6. La aplicación Flask se conecta a DynamoDB.

La implementación de esta opción fue, con diferencia, la más compleja del proyecto. Esta complejidad no provino del código de la aplicación, sino de la enrevesada configuración de red necesaria para conectar de forma segura el API Gateway (un servicio público) a un contenedor que se ejecuta en una subred privada.

La decisión de diseño principal fue priorizar la seguridad. El contenedor de Fargate no debía ser accesible desde la Internet pública. Para lograr esto, se implementó la siguiente arquitectura de red en la plantilla `03-ecs-option-a.yml`:

- Se provisionó un Network Load Balancer (NLB) con un esquema internal. Esto asegura que el balanceador solo tenga una dirección IP privada dentro de la VPC.
- Dado que el API Gateway no puede comunicarse directamente con una IP privada en una VPC, se hizo obligatorio el uso de un `VpcLink`. Este componente actúa como un túnel privado, permitiendo a API Gateway enrutar el tráfico de forma segura al NLB interno.

Una dificultad encontrada en la implementación de esta opción a mencionar es la siguiente:

Una configuración inicial con `AssignPublicIp: DISABLED` (para máxima seguridad) fallaba inmediatamente con un error de `ResourceInitializationError`. Los logs de ECS revelaron que la tarea no podía iniciarse porque era "unable to pull... registry auth". Al no tener IP pública y al estar en una subred sin un NAT Gateway, la tarea no tenía ninguna ruta de salida a Internet. Por lo tanto, no

podía contactar con la API de ECR para descargar su propia imagen. La solución fue establecer `AssignPublicIp: ENABLED` en la `NetworkConfiguration` del `ECSService`. Esto otorga a la tarea una IP pública efímera únicamente durante su fase de inicialización. La tarea usa esta IP para contactar ECR, descargar la imagen, y luego se registra en el NLB. Esta decisión no compromete la seguridad de la aplicación. La IP pública no se utiliza para recibir tráfico entrante de la aplicación, ya que el `ECSSecurityGroup` solo expone el puerto 8080 al NLB. El NLB sigue siendo internal y el único punto de entrada para los usuarios sigue siendo el API Gateway, protegido por la API Key.

OPCIÓN B: API DE MICROSERVICIOS CON AWS LAMBDA

Esta arquitectura implementa la API siguiendo el modelo FaaS (Función como Servicio), cumpliendo con el requisito de una solución desacoplada. En lugar de un monolito, la lógica de negocio se divide en 5 funciones independientes. Para evitar la duplicación de código, se creó una carpeta `app-lambda/shared/`. Esta carpeta contiene los modelos de validación de datos (`models.py`, usando Pydantic) y funciones de utilidad (`utils.py`).

El stack `04-lambda-option-b.yml` es notablemente más simple. El flujo de la petición es:

1. El usuario llama a API Gateway.
2. API Gateway invoca la función Lambda correspondiente (`AWS::Lambda::Function`) a través de una integración nativa.
3. La función Lambda se conecta a DynamoDB.

PUESTA EN MARCHA Y VERIFICACIÓN DE DESPLIEGUE

La arquitectura del proyecto, permite que el despliegue completo de cualquiera de las dos opciones (Fargate o Lambda) se realice de forma totalmente automatizada. La puesta en marcha no requiere ninguna configuración manual en la consola de AWS, en su lugar, se ejecuta una secuencia de scripts de Python mencionados anteriormente que son los que orquestan la creación de todos los recursos.

PRERREQUISITOS:

Antes de ejecutar los scripts de despliegue automatizado, es necesario asegurar que el entorno local del operador cumple con los siguientes requisitos técnicos, los cuales son indispensables para la correcta ejecución de los scripts de Python y la interacción con AWS.

PYTHON 3 Y PIP

Se requiere una instalación de Python 3.11+ (el proyecto se ha probado con Python 3.11). Python es el lenguaje en el que están escritos todos los scripts de automatización y la lógica de las aplicaciones. Se necesita pip para instalar las dependencias de los scripts y de las funciones Lambda.

BOTO3

Es necesario tener instalada la librería boto3, el SDK oficial de AWS para Python. Todos los scripts de despliegue la utilizan para interactuar con las APIs de AWS

DOCKER DESKTOP(SOLO PARA LA OPCIÓN A)

Para el despliegue de Fargate, es un requisito indispensable tener Docker Desktop (o un motor de Docker equivalente) en ejecución. El script push-image.py invoca directamente los comandos docker build y docker push para construir la imagen de la aplicación Flask y subirla al repositorio ECR.

AWS CLI Y CREDENCIALES CONFIGURADAS

Para que los scripts de Python (Boto3) y los comandos de Docker puedan interactuar con la API de AWS, es indispensable configurar la Interfaz de Línea de Comandos (CLI) de AWS en el entorno local. Esta configuración almacena de forma segura las credenciales que los scripts utilizarán para autenticarse.

Antes de configurar la CLI, se deben obtener las credenciales de un usuario de IAM. Estas credenciales consisten en un Access Key ID y una Secret Access Key. Estas credenciales, como se aprecia en la Figura1, junto con un token de sesión, son proporcionadas directamente por la interfaz del laboratorio.

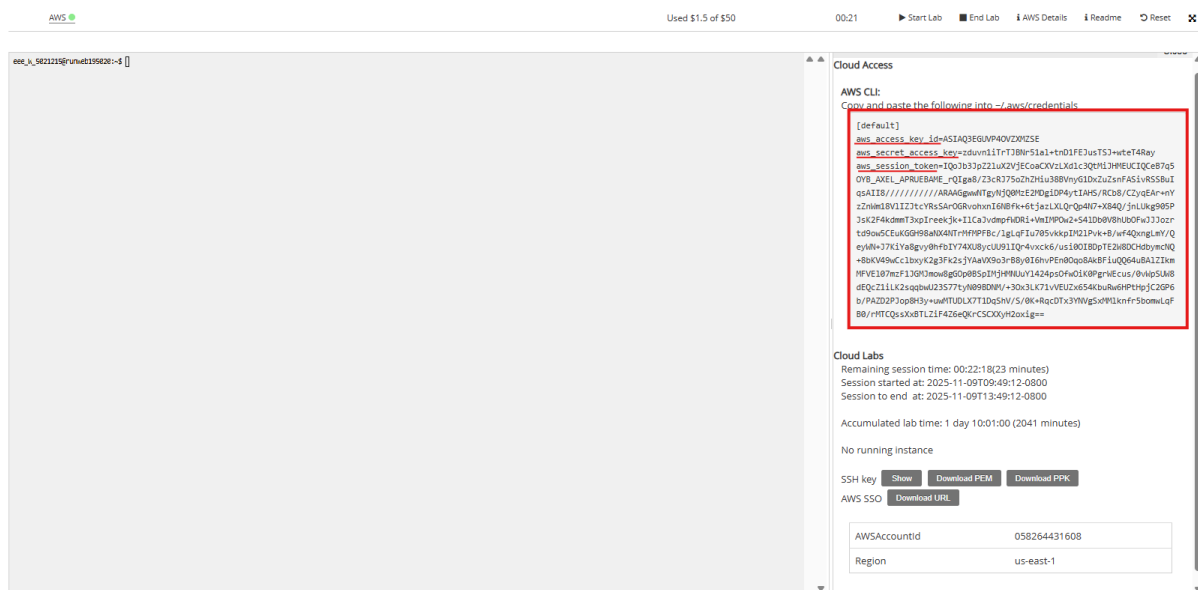


Figura1: Credenciales de AWS CLI

Una vez obtenidas las claves, se debe abrir una terminal CMD y ejecutar el comando **aws configure** como se aprecia en la Figura2. Este comando inicia un asistente interactivo que solicita cuatro datos:

1. AWS Access Key ID: Se pega el ID de la clave proporcionado por el laboratorio.
2. AWS Secret Access Key: Se pega la clave secreta.
3. Default region name: Se introduce la región donde se desplegará el proyecto. Para esta práctica, la región seleccionada es us-east-1.
4. Default output format: Se especifica el formato de salida para una mejor legibilidad como JSON o YAML

```
C:\Users\Dani\Desktop\CN_AWS_P1>aws configure
AWS Access Key ID [*****BYWB]: ASIAQ3EGUVP40VZXMZSE
AWS Secret Access Key [*****srnc]: zduvn1iTrTJBnr51al+tnD1FEJusTSJ+wteT4Ray
AWS Session Token [*****M7sw]: IQoJb3JpZ2luc2VjECoaCXVzLXdlc3QtMiJHMEUCIQCeB7q5OYBUQSFHuLSKjzLz49e67BCGzkRyDq
JNoW0zrQIga8/Z3cRj75oZhZHiu38BVnyG1DxZuZsnFASivRSSBuIqsAII8//////////ARAAGgwwNTgyNjQ0MzE2MDgiDP4ytIAHS/RCb8/CZyqEAR+nYz
ZnWm18VLIZJtcYRsSArOGRvohxnI6NBfk+6tjazLXLQrQp4N7+X84Q/jnLUkg905PJsK2F4kdmT3xpIreekjk+I1CaJvdmPfwDRi+VmIMPOw2+S4lDb0V8h
UboFwJJJozrtd9ow5CEuKGGH98aNX4NTrMFMPFBc/LgLfIu70SvkkpIM2LPvk+B/wf4QxngLmY/QeyWN+J7KiYa8gvy0hfbiY74XU8ycUU9LIQr4vxc6/u
si00IBDpTE2W8DCHdbymcNQ+8bKV49wCcLbxyK2g3Fk2sjYAaVX9o3rB8y0I6hvPEN0qo8AkBFiuQQ64uBALZIKmMFVEl07mzF1JGMJmow8gG0p0BSpIMjH
MNUuYlU24ps0fwOiK0PgrWEcus/0vWpSUW8dEQcZ1iLK2sqgbwU23S77tyN09BDNM/+30x3LK71vVEUZx654KbuRw6HPtHpjC2GP6b/PAZD2PJop8H3y+uwM
TUDLX7T1DqShV/S/0K+RqcDTx3YNVgSxMMLknfr5bomwLqFB0/rMTCQssXxBTLZiF4Z6eQKrCSCXXyH2oxig
Default region name [us-east-1]:
Default output format [json]:
```

Figura2: aws configure

Para confirmar que la CLI está configurada correctamente y que las credenciales son válidas y tienen permisos, se ejecuta un comando de verificación.

El comando estándar para esto es `aws sts get-caller-identity`. Este comando contacta con el servicio de seguridad de AWS (STS) y devuelve la identidad (el ARN) del usuario o rol que la CLI está utilizando. Si la configuración es exitosa se debe ver algo parecido a la Figura3.

```
C:\Users\Dani\Desktop\CN_AWS_P1>aws sts get-caller-identity
{
  "UserId": "AROAQ3EGUVP4NBVCQYBHJ:user4384698=daniel.moreno",
  "Account": "058264431608",
  "Arn": "arn:aws:sts::058264431608:assumed-role/voclabs/user4384698=daniel.moreno"
}

C:\Users\Dani\Desktop\CN_AWS_P1>
```

Figura3: Comprobacion exitosa Aws

DESPLIEGUE AUTOMATIZADO

Todos los scripts usados a continuación pueden encontrarse en la carpeta scripts/ dentro del proyecto.

DESPLIEGUE ECS FARGATE

La puesta en marcha de la arquitectura de contenedores requiere cuatro pasos secuenciales una vez nos hemos movido a la carpeta base del proyecto:

1. CREAR LA BASE DE DATOS

Con el script ***python scripts/deploy-dynamodb.py*** como se ve en la Figura4 creamos el stack notes-dynamodb, que crea la tabla Notes en DynamoDB y nos devuelve tanto el DynamoDBTableName que es el nombre de la tabla creada como el DynamoDBTableARN que es su identificador único en AWS.

```
C:\Users\Dani\Desktop\CN_AWS_P1>python scripts/deploy-dynamodb.py
Desplegando tabla DynamoDB...
Stack: notes-dynamodb
Region: us-east-1

Validando template...
Template válido

Creando stack...
Esperando a que se complete...
Stack creado exitosamente!

Outputs:
  TableName: Notes
  TableArn: arn:aws:dynamodb:us-east-1:058264431608:table/Notes

C:\Users\Dani\Desktop\CN_AWS_P1>
```

Figura4: Despliegue DynamoDB

2. CREAR EL REGISTRO DE CONTENEDORES

Con el script python ***scripts/deploy-ecr.py*** creamos el stack notes-ecr, que crea el repositorio ECR donde se aloja la imagen Docker. Como se comprueba en la Figura5, se crea exitosamente con tanto el nombre, el Arn y el RepositoryUri el cual es la URL completa que el script push-image.py usará como destino para subir la imagen de Docker. Además se añaden los siguientes pasos para terminar de desplegar.

```
C:\Users\Dani\Desktop\CN_AWS_P1>python scripts/deploy-ecr.py
Desplegando repositorio ECR...
Stack: notes-ecr
Region: us-east-1

Validando template...
Template válido

Creando stack...
Esperando a que se complete...
Stack creado exitosamente!

Outputs:
  RepositoryName: notes-app
  RepositoryArn: arn:aws:ecr:us-east-1:058264431608:repository/notes-app
  RepositoryUri: 058264431608.dkr.ecr.us-east-1.amazonaws.com/notes-app

Próximo paso:
  1. Construir imagen Docker: cd app-ecs && docker build -t notes-app .
  2. Subir imagen: python scripts/push-image.py

C:\Users\Dani\Desktop\CN_AWS_P1>
```

Figura5: Despliegue ECR

3. CONSTRUIR Y SUBIR LA IMAGEN DE DOCKER

Con los comandos vistos antes y el script ***python scripts/push-image.py*** se ejecuta el DockerFile de app-ecs/ , se construye la imagen de la aplicación Flask, se autentica y se sube la imagen al repositorio, tal y como apreciamos en las Figura6 , Figura7 y Figura8.

```
C:\Users\Dani\Desktop\CM_AWS_P1>cd app-ecs

C:\Users\Dani\Desktop\CM_AWS_P1>app-ecs>docker build -t notes-app .
[+] Building 3.4s (11/11) FINISHED
    docker:desktop-linux
    [internal] load build definition from Dockerfile 0.0s
    ==> transferring dockerfile: 331B 0.0s
    [internal] load metadata for docker.io/library/python:3.11-slim 2.9s
    ==> CACHED [2/5] FROM docker.io/library/python:3.11-slim@sha256:4e670727fba8392e5cd8ba85262b4d9e9e6ebd055740d93e361 0.0s
    [internal] load build context 0.0s
    ==> transferring context: 157B 0.0s
    [internal] load build context 0.0s
    ==> transferring context: 64B 0.0s
    [1/5] FROM docker.io/library/python:3.11-slim@sha256:4e670727fba8392e5cd8ba85262b4d9e9e6ebd055740d93e361 0.0s
    ==> CACHED [2/5] FROM docker.io/library/python:3.11-slim@sha256:4e670727fba8392e5cd8ba85262b4d9e9e6ebd055740d93e361 0.0s
    CACHED [2/5] WORKDIR /app 0.0s
    CACHED [3/5] COPY requirements.txt 0.0s
    CACHED [4/5] RUN pip install --no-cache-dir -r requirements.txt 0.0s
    CACHED [5/5] COPY main.py 0.0s
    ==> exporting to image 0.2s
    ==> exporting manifest sha256:baceb070f148621552329013f4e8972c40f6d40f1a9c79533c4bd4e4f 0.0s
    ==> exporting config sha256:b76f7d3b302c2388b4aa826893faeaa5540b195747d3bb48a18910215849 0.0s
    ==> exporting attestation manifest sha256:b9d3317a247847927284049bf0494b5cf4f63bd6562624d29a4c76f7b373 0.1s
    ==> exporting manifest list sha256:b2b0b1b3b0b0d6f27b3624b34e31a1c1b9817cadedf14c23eeef83eb1f9569 0.0s
    ==> naming to docker.io/library/notes-app:latest 0.0s
    ==> unpacking to docker.io/library/notes-app:latest 0.0s

C:\Users\Dani\Desktop\CM_AWS_P1>app-ecs>cd ..

C:\Users\Dani\Desktop\CM_AWS_P1>python scripts/push-image.py
Construyendo y subiendo imagen Docker a ECR...

Repositorio ECR: 858264u31688.dkr.ecr.us-east-1.amazonaws.com/notes-app

Autenticando con ECR...
[Ejecute: echo $AWS_SECRET_KEY | jq -r .secret | xargs aws ecr get-login-password --profile default --region us-east-1 | xargs docker login --username AWS --password stdin https://858264u31688.dkr.ecr.us-east-1.amazonaws.com/]
login Succeeded

Autenticación exitosa

Construyendo imagen Docker...
[Ejecute: docker build -t 858264u31688.dkr.ecr.us-east-1.amazonaws.com/notes-app:latest app-ecs/]
[+] Building 3.4s (11/11) FINISHED
    docker:desktop-linux
    [internal] load build definition from Dockerfile 0.0s
    ==> transferring dockerfile: 331B done 0.0s
    [internal] load metadata for docker.io/library/python:3.11-slim 2.9s
    ==> CACHED [2/5] FROM docker.io/library/python:3.11-slim@sha256:4e670727fba8392e5cd8ba85262b4d9e9e6ebd055740d93e361 0.0s
    [internal] load build context 0.0s
    ==> transferring context: 157B done 0.0s
    [internal] load build context 0.0s
    ==> transferring context: 64B done 0.0s
    [1/5] FROM docker.io/library/python:3.11-slim@sha256:4e670727fba8392e5cd8ba85262b4d9e9e6ebd055740d93e361 0.0s
    ==> CACHED [2/5] FROM docker.io/library/python:3.11-slim@sha256:4e670727fba8392e5cd8ba85262b4d9e9e6ebd055740d93e361 0.0s
    CACHED [2/5] WORKDIR /app 0.0s
    CACHED [3/5] COPY requirements.txt 0.0s
    CACHED [4/5] RUN pip install --no-cache-dir -r requirements.txt 0.0s
    CACHED [5/5] COPY main.py 0.0s
    ==> exporting to image 0.2s
    ==> exporting manifest sha256:baceb070f148621552329013f4e8972c40f6d40f1a9c79533c4bd4e4f 0.0s
    ==> exporting config sha256:b76f7d3b302c2388b4aa826893faeaa5540b195747d3bb48a18910215849 0.0s
    ==> exporting attestation manifest sha256:b9d3317a247847927284049bf0494b5cf4f63bd6562624d29a4c76f7b373 0.1s
    ==> exporting manifest list sha256:b2b0b1b3b0b0d6f27b3624b34e31a1c1b9817cadedf14c23eeef83eb1f9569 0.0s
    ==> naming to docker.io/library/notes-app:latest 0.0s
    ==> unpacking to docker.io/library/notes-app:latest 0.0s
```

Figura6: Construcccion del Docker

```

#4 [internal] load build context
#4 transferring context: 64B done
#4 DONE 0.0s

#5 [1/5] FROM docker.io/library/python:3.11-slim@sha256:e4676722fba839e2e5cdb844a52262b43e90e56dbd55b7ad953ee3615ad7534f
#5 resolve docker.io/library/python:3.11-slim@sha256:e4676722fba839e2e5cdb844a52262b43e90e56dbd55b7ad953ee3615ad7534f 0.0s done
#5 DONE 0.0s

#6 [2/5] WORKDIR /app
#6 CACHED

#7 [4/5] RUN pip install --no-cache-dir -r requirements.txt
#7 CACHED

#8 [3/5] COPY requirements.txt .
#8 CACHED

#9 [5/5] COPY main.py .
#9 CACHED

#10 exporting to image
#10 exporting layers done
#10 exporting manifest sha256:8acee0a70f1689621552d302012f3a8027ca94d8664f41e99af7953c4bda646f done
#10 exporting config sha256:b7df25bb202e2380b64aea826885fefaa955c0bb195736db86a01b0510258cd9 done
#10 exporting attestation manifest sha256:4650fa1d82dc39d41d1e603b89bc36bd54f46d07223c12344c3cfa9157db5d2a
#10 exporting attestation manifest sha256:4650fa1d82dc39d41d1e603b89bc36bd54f46d07223c12344c3cfa9157db5d2a 0.0s done
#10 exporting manifest list sha256:51a651e2eb497dc7a0fef6a1b56e066b8c2a4f7da40a46c491eeelccd155368b 0.0s done
#10 naming to 058264431608.dkr.ecr.us-east-1.amazonaws.com/notes-app:latest done
#10 unpacking to 058264431608.dkr.ecr.us-east-1.amazonaws.com/notes-app:latest 0.0s done
#10 DONE 0.2s

Imagen construida exitosamente

Subiendo imagen a ECR...
Ejecutando: docker push 058264431608.dkr.ecr.us-east-1.amazonaws.com/notes-app:latest
The push refers to repository [058264431608.dkr.ecr.us-east-1.amazonaws.com/notes-app]
aae2c7de8ea6: Waiting
d7ecdcd7702a: Waiting
023637af2351: Waiting
ed3c84527d3f: Waiting
f002d17b63fe: Waiting
65868b001a40: Waiting
1ee9c106547f: Waiting
935ecc164f86: Waiting
3294992f5df7: Waiting
3294992f5df7: Waiting
aae2c7de8ea6: Waiting
d7ecdcd7702a: Waiting
023637af2351: Waiting
ed3c84527d3f: Waiting
f002d17b63fe: Waiting
65868b001a40: Waiting
1ee9c106547f: Waiting
935ecc164f86: Waiting
f002d17b63fe: Waiting
65868b001a40: Waiting
1ee9c106547f: Waiting
935ecc164f86: Waiting
3294992f5df7: Waiting
aae2c7de8ea6: Waiting
d7ecdcd7702a: Waiting
023637af2351: Waiting
ed3c84527d3f: Waiting

```

Figura7: Construcción del Docker

```

023637af2351: Waiting
ed3c84527d3f: Waiting
f002d17b63fe: Waiting
65868b001a40: Waiting
1ee9c106547f: Waiting
935ecc164f86: Waiting
3294992f5df7: Waiting
aae2c7de8ea6: Waiting
d7ecded7702a: Waiting
023637af2351: Waiting
3294992f5df7: Waiting
aae2c7de8ea6: Waiting
d7ecded7702a: Waiting
023637af2351: Waiting
ed3c84527d3f: Waiting
f002d17b63fe: Waiting
65868b001a40: Waiting
1ee9c106547f: Waiting
935ecc164f86: Waiting
3294992f5df7: Waiting
aae2c7de8ea6: Waiting
d7ecded7702a: Waiting
023637af2351: Waiting
ed3c84527d3f: Waiting
f002d17b63fe: Waiting
65868b001a40: Waiting
1ee9c106547f: Waiting
935ecc164f86: Waiting
3294992f5df7: Waiting
aae2c7de8ea6: Waiting
d7ecded7702a: Waiting
023637af2351: Waiting
935ecc164f86: Waiting
d7ecded7702a: Waiting
023637af2351: Waiting
65868b001a40: Waiting
ed3c84527d3f: Pushed
935ecc164f86: Pushed
3294992f5df7: Pushed
aae2c7de8ea6: Pushed
65868b001a40: Pushed
1ee9c106547f: Pushed
f002d17b63fe: Pushed
d7ecded7702a: Pushed
023637af2351: Pushed
latest: digest: sha256:51a651e2eb497dc7a0fef6a1b56e066b8c2a4f7da40a46c491eee1ccd155368b size: 856

Imagen subida exitosamente!
URI completa: 058264431608.dkr.ecr.us-east-1.amazonaws.com/notes-app:latest
C:\Users\Dani\Desktop\CN_AWS_P1>

```

Figura8: Subida Imagen

4. DESPLEGAR LA API DE FARGATE

python scripts/deploy-ecs.py es el script principal, el cual lee la plantilla 03-ecs-option-a.yml, crea el stage dinámico y despliega todos los recursos de la Opción A: el clúster de ECS, el NLB, el VpcLink, el API Gateway y el servicio de Fargate. Al finalizar, la terminal muestra la ApiUrl y la ApiKey necesarias para la prueba como se comprueba en la Figura9.

```

C:\Users\Dani\Desktop\CN_AWS_P1>python scripts/deploy-ecs.py
Desplegando ECS Fargate (Opción A)...
Stack: notes-ecs-option-a
Region: us-east-1

Obteniendo VPC y subnets...
VPC: vpc-0ad52cfe9e8f15ad8
Subnets: subnet-0d5f554cfe28874a1, subnet-0860ebab9ce4542a7

Obteniendo imagen ECR...
Imagen: 058264431608.dkr.ecr.us-east-1.amazonaws.com/notes-app:latest

Validando template...
Template válido

Desplegando en Stage: v1762727079

Stage name: v1762727079
Creando stack (esto puede tardar 5-10 minutos)...
Esperando a que se complete...
Stack creado exitosamente!

=====
INFORMACIÓN IMPORTANTE
=====
ApiUrl: https://wldxylical.execute-api.us-east-1.amazonaws.com/v1762727079
ApiKey: 2h6m3u3wpg
ServiceName: notes-service
NLBDnsName: notes-nlb-e51748ffd666bebf.elb.us-east-1.amazonaws.com
ClusterName: notes-cluster

=====
PRUEBA LA API
=====

URL: https://wldxylical.execute-api.us-east-1.amazonaws.com/v1762727079
API Key: yzhIM1EBY34CUKluDKTm37NuI8SHp9w82QNgA9KO

Ejemplo:
curl -X POST https://wldxylical.execute-api.us-east-1.amazonaws.com/v1762727079/notes \
-H "x-api-key: yzhIM1EBY34CUKluDKTm37NuI8SHp9w82QNgA9KO" \
-H "Content-Type: application/json" \
-d '{"title": "Test", "content": "Prueba ECS", "tags": ["test"]}'

C:\Users\Dani\Desktop\CN_AWS_P1>

```

Figura9: Despliegue ECS

DESPLIEGUE AWS LAMBDA

La puesta en marcha de la arquitectura de microservicios requiere tres pasos secuenciales una vez nos hemos movido a la carpeta base del proyecto:

1. CREAR LA BASE DE DATOS

Este paso es idéntico al despliegue con Ecs Fargate, en caso de ya haberlo realizado, este paso debe omitirse ya que la tabla es compartida. En caso de solo querer tener el despliegue con Lambda se seguirá el mismo método mencionado anteriormente y también mostrado en la Figura4.

2. EMPAQUETAR LAS FUNCIONES LAMBDA

El script ***python scripts/package-lambdas.py*** es crucial. Itera sobre cada función en app-lambda/, instala sus dependencias (requirements.txt), y empaqueta el handler.py junto con la carpeta shared/ en archivos .zip listos para el despliegue. Como se demuestra en la Figura10, la terminal confirma que el script itera por cada una de las 5 funciones, instala sus dependencias y empaqueta el handler.py junto con el código shared/ (que contiene los modelos). El resultado son los 5 archivos .zip autocontenidos, listos para ser desplegados. Durante la instalación, se observa un aviso que sugiere actualizar pip. Esta advertencia es puramente informativa y no afecta el éxito del despliegue, por lo que puede ser ignorada.

```

C:\Users\Dani\Desktop\CN_AWS_P1>python scripts/package-lambdas.py
=====
EMPAQUETADO DE FUNCIONES LAMBDA
=====

Directorio lambda-packages/ creado

Empaquetando create_note...
  Instalando dependencias...

[notice] A new release of pip is available: 25.2 -> 25.3
[notice] To update, run: python.exe -m pip install --upgrade pip
✓ Creado: lambda-packages\create_note.zip (14.20 MB)

Empaquetando get_note...
  Instalando dependencias...

[notice] A new release of pip is available: 25.2 -> 25.3
[notice] To update, run: python.exe -m pip install --upgrade pip
✓ Creado: lambda-packages\get_note.zip (14.20 MB)

Empaquetando list_notes...
  Instalando dependencias...

[notice] A new release of pip is available: 25.2 -> 25.3
[notice] To update, run: python.exe -m pip install --upgrade pip
✓ Creado: lambda-packages\list_notes.zip (14.20 MB)

Empaquetando update_note...
  Instalando dependencias...

[notice] A new release of pip is available: 25.2 -> 25.3
[notice] To update, run: python.exe -m pip install --upgrade pip
✓ Creado: lambda-packages\update_note.zip (14.20 MB)

Empaquetando delete_note...
  Instalando dependencias...

[notice] A new release of pip is available: 25.2 -> 25.3
[notice] To update, run: python.exe -m pip install --upgrade pip
✓ Creado: lambda-packages\delete_note.zip (14.20 MB)

=====
EMPAQUETADO COMPLETADO
=====

Archivos ZIP creados en: lambda-packages/

Próximo paso:
python scripts/deploy-lambda.py

C:\Users\Dani\Desktop\CN_AWS_P1>

```

Figura10: Empaquetado de lambdas

3. DESPLEGAR LA API DE LAMBDA

Con el script `python scripts/deploy-lambda.py` se despliega el stack `04-lambda-option-b.yml`. Este despliegue se realiza en dos fases: primero, CloudFormation crea el stack y segundo, el propio script sube los `.zip` al bucket S3 y actualiza el código de las funciones Lambda. Al finalizar, la terminal muestra la `ApiUrl` y la `ApiKey` correspondientes a la Opción B como se ve en la Figura11.

```
C:\Users\Dani\Desktop\CN_AWS_P1>python scripts/deploy-lambda.py
=====
DESPLIEGUE DE LAMBDA (OPCIÓN B)
=====

✓ Todos los paquetes Lambda encontrados

Desplegando funciones Lambda (Opción B)...
Stack: notes-lambda-option-b
Region: us-east-1

Validando template...
✓ Template válido

Obteniendo Account ID de AWS...
✓ Account ID: 058264431608

Creando stack (esto puede tardar 3-5 minutos)...
Esperando creación del stack...
✓ Stack creado exitosamente!

=====
INFORMACIÓN IMPORTANTE
=====

ApiUrl: https://uoubqhzq31.execute-api.us-east-1.amazonaws.com/prod
ApiKey: wcr8irn3hi
ListNotesFunctionArn: arn:aws:lambda:us-east-1:058264431608:function:ListNotesFunction
UpdateNoteFunctionArn: arn:aws:lambda:us-east-1:058264431608:function:UpdateNoteFunction
DeleteNoteFunctionArn: arn:aws:lambda:us-east-1:058264431608:function:DeleteNoteFunction
DeploymentBucket: notes-lambda-deployment-058264431608
CreateNoteFunctionArn: arn:aws:lambda:us-east-1:058264431608:function:CreateNoteFunction
GetNoteFunctionArn: arn:aws:lambda:us-east-1:058264431608:function:GetNoteFunction

=====
SUBIENDO CÓDIGO LAMBDA
=====

Subiendo paquetes Lambda a S3...
Subiendo create_note.zip...
✓ functions/create_note.zip
Subiendo get_note.zip...
✓ functions/get_note.zip
Subiendo list_notes.zip...
✓ functions/list_notes.zip
Subiendo update_note.zip...
✓ functions/update_note.zip
Subiendo delete_note.zip...
✓ functions/delete_note.zip

✓ Todos los paquetes subidos

Actualizando código de las funciones Lambda...
Actualizando CreateNoteFunction...
Actualizando GetNoteFunction...
Actualizando ListNotesFunction...
Actualizando UpdateNoteFunction...
Actualizando DeleteNoteFunction...

✓ Código actualizado en todas las funciones

=====
PRUEBA LA API
=====

URL: https://uoubqhzq31.execute-api.us-east-1.amazonaws.com/prod
API Key: be8f21atxf9A1zmEkwiFz2Vto06oLRGm3CgpbLRu

Ejemplo:
curl -X POST https://uoubqhzq31.execute-api.us-east-1.amazonaws.com/prod/notes \
-H "x-api-key: be8f21atxf9A1zmEkwiFz2Vto06oLRGm3CgpbLRu" \
-H "Content-Type: application/json" \
-d '{"title": "Test Lambda", "content": "Desde Lambda!", "tags": ["Lambda"]}'

C:\Users\Dani\Desktop\CN_AWS_P1>
```

Figura11: Despliegue Lambdas

VERIFICACIÓN EN LA CONSOLA DE AWS

Tras la ejecución de los scripts, se puede verificar la creación exitosa de todos los recursos directamente en la consola de AWS, validando el éxito de la Infraestructura como Código.

STACKS DE CLOUDFORMATION

El servicio AWS CloudFormation actúa como el "panel de control" central de nuestra infraestructura. Al navegar a este servicio, se pueden observar todos los stacks que nuestros scripts han creado.

La verificación clave es que el Estado de todos los stacks sea CREATE_COMPLETE. Esto confirma que AWS ha provisionado con éxito todos los recursos definidos en nuestras plantillas YAML.

Como vemos en la Figura12, todos los scripts anteriores aparecen con dicho estado y listo para funcionar.


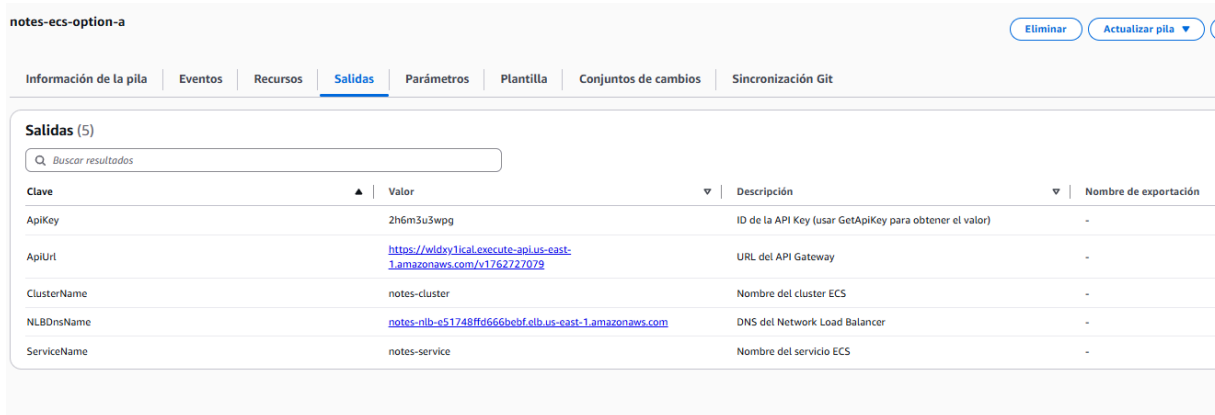
Pilas (5)					 Eliminar
<input type="text" value="Buscar por nombre de pila"/>					Filtrar estado
					Activo
					<input checked="" type="checkbox"/> Vista anidada
	Nombre de la pila	Estado	Hora de creación	Descripción	
<input type="radio"/>	notes-lambda-option-b	CREATE_COMPLETE	2025-11-09 23:47:55 UTC+0000	Opcion B - Lambda con API Gateway para aplicacion de notas	
<input type="radio"/>	notes-ecs-option-a	CREATE_COMPLETE	2025-11-09 22:24:45 UTC+0000	ECS Fargate con API Gateway (Opcion A) - Optimizado	
<input type="radio"/>	notes-ecr	CREATE_COMPLETE	2025-11-09 22:01:59 UTC+0000	Repositorio ECR para la aplicacion de notas	
<input type="radio"/>	notes-dynamodb	CREATE_COMPLETE	2025-11-09 21:46:26 UTC+0000	Tabla DynamoDB para la aplicacion de notas	
<input type="radio"/>	c177090a45702621115377331w058264431608	CREATE_COMPLETE	2025-09-10 19:31:53 UTC+0100	associate Learner Lab template (academy)	

Figura12: Panel CloudFormation

SALIDAS DE LOS STACKS

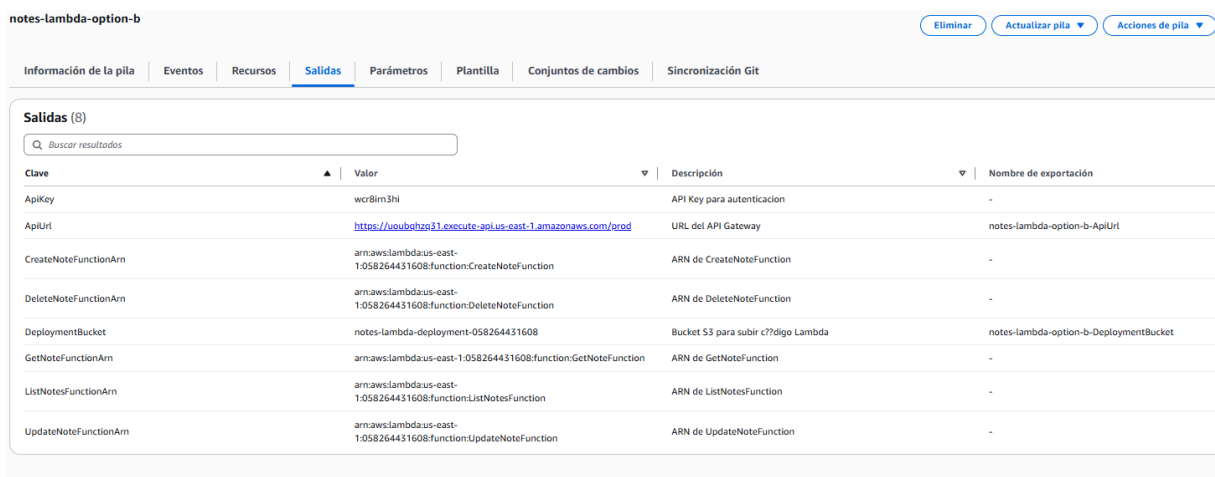
Cada stack de despliegue ha sido configurado para exportar las variables importantes, como la URL de la API. Estas salidas se pueden inspeccionar en la consola de AWS.

Al seleccionar un stack y hacer clic en la pestaña "Salidas", se pueden ver los valores generados por el despliegue, como la ApiUrl y el ApiKey. Esto confirma que CloudFormation ha conectado y provisionado los recursos correctamente, y nos proporciona los puntos de acceso que coinciden con la salida de nuestro script, tal y como se comprueba en las Figura13 y Figura14.



Clave	Valor	Descripción	Nombre de exportación
ApiKey	2h6m3u3wpq	ID de la API Key (usar GetApiKey para obtener el valor)	-
ApiUrl	https://wldxy1lcal.execute-api.us-east-1.amazonaws.com/v1762727079	URL del API Gateway	-
ClusterName	notes-cluster	Nombre del cluster ECS	-
NLBName	notes-nlb-s51748ff66656bcbf.elb.us-east-1.amazonaws.com	DNS del Network Load Balancer	-
ServiceName	notes-service	Nombre del servicio ECS	-

Figura13: Salida CloudFormation notes-ecs-option-a



Clave	Valor	Descripción	Nombre de exportación
ApiKey	wcr8im3hi	API Key para autentificación	-
ApiUrl	https://uoubohzo31.execute-api.us-east-1.amazonaws.com/prod	URL del API Gateway	notes-lambda-option-b-ApiUrl
CreateNoteFunctionArn	arn:aws:lambda:us-east-1:058264431608:function:CreateNoteFunction	ARN de CreateNoteFunction	-
DeleteNoteFunctionArn	arn:aws:lambda:us-east-1:058264431608:function:DeleteNoteFunction	ARN de DeleteNoteFunction	-
DeploymentBucket	notes-lambda-deployment-058264431608	Bucket S3 para subir código Lambda	notes-lambda-option-b-DeploymentBucket
GetNoteFunctionArn	arn:aws:lambda:us-east-1:058264431608:function:GetNoteFunction	ARN de GetNoteFunction	-
ListNotesFunctionArn	arn:aws:lambda:us-east-1:058264431608:function:ListNotesFunction	ARN de ListNotesFunction	-
UpdateNoteFunctionArn	arn:aws:lambda:us-east-1:058264431608:function:UpdateNoteFunction	ARN de UpdateNoteFunction	-

Figura14: Salida CloudFormation notes-lambda-option-b

PRUEBAS Y VERIFICACIÓN

Una vez completados los despliegues de las dos arquitecturas, se procederá a una fase de verificación funcional. El objetivo de esta fase es doble:

1. Validar que ambas implementaciones de backend cumplieran con el contrato de la API (las cinco operaciones CRUD).
2. Demostrar que ambas arquitecturas eran funcionalmente idénticas desde la perspectiva del cliente, tal como lo requería el diseño desacoplado.

Para esta verificación se utilizaron dos métodos complementarios, de acuerdo con los entregables de la práctica: un cliente de API (Postman) para pruebas directas de los *endpoints*, y una interfaz web rudimentaria para la validación funcional.

VERIFICACIÓN ENDPOINTS CON POSTMAN

Se utiliza la herramienta Postman para realizar pruebas directas contra los endpoints de API Gateway. Se creó una colección de Postman que contenía las cinco peticiones HTTP (POST, GET, GET por ID, PUT, DELETE).

Para facilitar la conmutación entre ambas opciones, se configuraron variables de colección en Postman ({{baseUrl}} y {{api_key}}). El proceso de prueba fue el siguiente:

1. Se ejecutó el script de despliegue (ej. python scripts/deploy-lambda.py).
2. Se copiaron los valores ApiUrl y API Key (el valor secreto) de la salida de la terminal.
3. Estos valores se pegaron en las variables de la colección de Postman, permitiendo que las 5 peticiones apuntaran al backend deseado.

PRUEBA CREACION(POST /notes)

The screenshot shows the Postman interface for a POST request to the endpoint `[[api_url]]/notes`. The request is successful, returning a 201 status code. The response body is a JSON object representing a newly created note.

Request:

- Method: POST
- URL: `[[api_url]]/notes`

Response (201 Created):

```
{
  "note_id": "e26e680e-4901-40f5-be1e-1af93bf783ab",
  "title": "Nota de prueba desde Postman - 2025-11-10T00:31:50.761Z",
  "content": "Esta nota fue creada autom\u00e1ticamente por la colecci\u00f3n de Postman para probar el endpoint de creaci\u00f3n.",
  "tags": [
    "postman",
    "test",
    "automated"
  ],
  "created_at": "2025-11-10T00:31:56.561153Z",
  "updated_at": "2025-11-10T00:31:56.561153Z"
}
```

PRUEBA DE LECTURA (GET /notes)

The screenshot shows the Postman interface for a GET request to the endpoint `[[api_url]]/notes`. The request is successful, returning a 200 status code. The response body is a JSON array containing two note objects.

Request:

- Method: GET
- URL: `[[api_url]]/notes`

Response (200 OK):

```
[
  {
    "updated_at": "2025-11-10T00:31:56.561153Z",
    "content": "Esta nota fue creada autom\u00e1ticamente por la colecci\u00f3n de Postman para probar el endpoint de creaci\u00f3n.",
    "created_at": "2025-11-10T00:31:56.561153Z",
    "note_id": "e26e680e-4901-40f5-be1e-1af93bf783ab",
    "tags": [
      "postman",
      "test",
      "automated"
    ],
    "title": "Nota de prueba desde Postman - 2025-11-10T00:31:50.761Z"
  },
  {
    "updated_at": "2025-11-10T00:28:14.051372Z",
    "content": "1",
    "created_at": "2025-11-10T00:28:14.051372Z",
    "note_id": "3e32810f-66d3-4a01-9bc7-4d9d7aa7bb2a",
    "tags": [],
    "title": "Test"
  }
]
```

PRUEBA DE ACTUALIZACIÓN(PUT /notes/id)

POST Create NoteGET List All NotesPUT Update Note

Notes API - AWS Practice / Update Note

PUT{{apiUrl}}/notes/{{note_id}}

Send

ParamsAuthorizationHeaders (11)BodyScriptsSettingsCookies

Query Params

Key	Value	Description
Key	Value	Description

BodyCookiesHeaders (10)Test Results (7/7)200 OK · 299 ms · 791 BSave Response

JSONPreviewVisualize

```
1 {
2   "content": "El contenido también fue actualizado para probar el endpoint PUT.",
3   "updated_at": "2025-11-10T00:35:36.847163Z",
4   "created_at": "2025-11-10T00:31:56.561153Z",
5   "note_id": "e26e650e-4901-48f5-be1e-1af93bf703ab",
6   "tags": [
7     "postman",
8     "test",
9     "updated"
10  ],
11   "title": "Nota ACTUALIZADA desde Postman"
12 }
```

POST Create NoteGET List All NotesPUT Update Note

Notes API - AWS Practice / List All Notes

GET{{apiUrl}}/notes

Send

ParamsAuthorizationHeaders (8)BodyScriptsSettingsCookies

Query Params

Key	Value	Description
Key	Value	Description

BodyCookiesHeaders (10)Test Results (8/8)200 OK · 296 ms · 981 BSave Response

JSONPreviewVisualize

```
1 [
2   {
3     "content": "El contenido también fue actualizado para probar el endpoint PUT.",
4     "updated_at": "2025-11-10T00:35:36.847163Z",
5     "created_at": "2025-11-10T00:31:56.561153Z",
6     "note_id": "e26e650e-4901-48f5-be1e-1af93bf703ab",
7     "tags": [
8       "postman",
9       "test",
10      "updated"
11    ],
12     "title": "Nota ACTUALIZADA desde Postman"
13  },
14  {
15    "updated_at": "2025-11-10T00:28:14.051372Z",
16    "content": "1",
17    "created_at": "2025-11-10T00:28:14.051372Z",
18    "note_id": "3e32810f-66d3-4a01-9bc7-4d9d7aa7bb2a",
19    "tags": [],
20    "title": "Test"
21  }
22 ]
```

PRUEBA DE BORRADO(DELETE /notes/id)

The screenshot shows a REST client interface with the following details:

- Method:** DELETE
- URL:** `{{apiUrl}}/notes/{{note_id}}`
- Params:** Query Params table with columns Key, Value, and Description.
- Body:** Empty
- Response:** 204 No Content, 1.61 s, 488 B
- Test Results:** 4/5

Key	Value	Description
Key	Value	Description

```
1
```

The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** `{{apiUrl}}/notes`
- Params:** Query Params table with columns Key, Value, and Description.
- Body:** JSON
- Response:** 200 OK, 224 ms, 670 B
- Test Results:** 7/8

Key	Value	Description
Key	Value	Description

```
1 [
2   {
3     "updated_at": "2025-11-10T00:28:14.051372Z",
4     "content": "1",
5     "created_at": "2025-11-10T00:28:14.051372Z",
6     "note_id": "3e32816f-66d3-4a81-9bc7-4d9d7aa7bb2a",
7     "tags": [],
8     "title": "Test"
9   }
10 ]
```

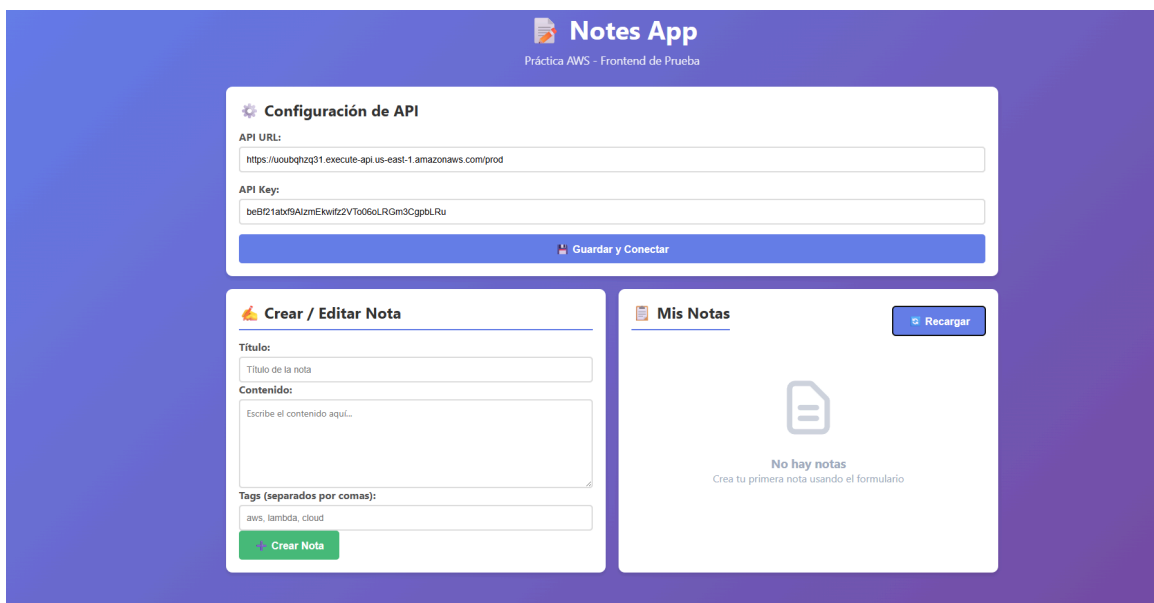
VERIFICACIÓN FUNCIONAL CON FRONTEND

Para cumplir con el requisito de la práctica de "preparar alguna interfaz rudimentaria de prueba", se utilizó el archivo notes-frontend.html proporcionado en el proyecto.

Este archivo se abre directamente en un navegador web y actúa como un cliente completo para la API.

CONFIGURACIÓN DEL CLIENTE WEB

El primer paso fue conectar el frontend con el backend desplegado. Para ello, se utilizaron los mismos valores de ApiUrl y ApiKey obtenidos de la salida del script de despliegue como se observa en la Figura15.



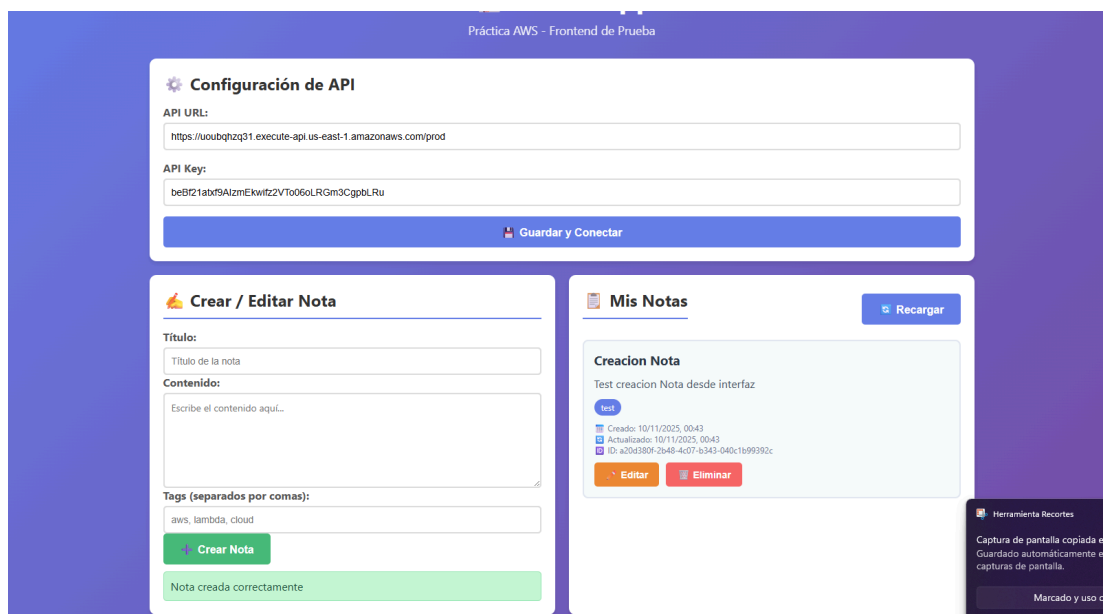
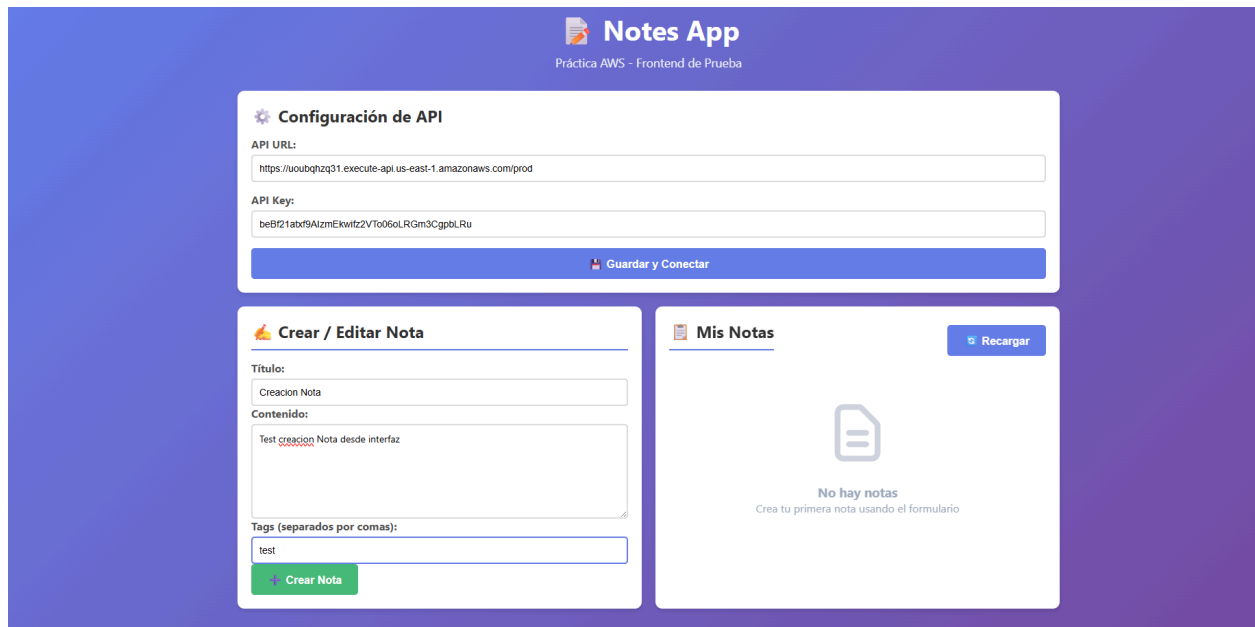
The screenshot displays the 'Notes App' interface with a purple header. Below the header, there's a 'Configuración de API' section with two input fields: 'API URL' containing 'https://ucubghzq31.execute-api.us-east-1.amazonaws.com/prod' and 'API Key' containing 'beBf21atx9AlzmEkwtZ2VTo0oLRGm3CgpbLRu'. A blue button labeled 'Guardar y Conectar' is below these fields. To the left, the 'Crear / Editar Nota' section has a 'Titulo:' field, a 'Contenido:' text area, and a 'Tags (separados por comas):' field with 'aws, lambda, cloud' entered. A green 'Crear Nota' button is at the bottom. To the right, the 'Mis Notas' section shows a document icon and the text 'No hay notas. Crea tu primera nota usando el formulario'. A blue 'Recargar' button is in the top right of this section.

Figura15: Inserción de credenciales

PRUEBA FUNCIONALIDAD CRUD

Tras guardar la configuración, la página ejecutó automáticamente una llamada GET /notes y pobló el panel con los datos de DynamoDB.

En el siguiente repertorio de imágenes se puede comprobar las funcionalidades CRUD.




Configuración de API

API URL:

https://uoubqhzq31.execute-api.us-east-1.amazonaws.com/prod

API Key:

beBf21atY9AizmEkwlz2VT006oLRGm3CgpbLRu

 Guardar y Conectar

Crear / Editar Nota

Título:

Creacion Nota(Ahora editada)

Contenido:

Test creacion Nota desde interfaz. espero aprobar

Tags (separados por comas):

test, imagen



Actualizar Nota



Cancelar

Mis Notas

 Recargar

Creacion Nota

Test creacion Nota desde interfaz

test

 Creado: 10/11/2025, 00:43
 Actualizado: 10/11/2025, 00:43
 ID: a20d380f-2b48-4c07-b343-040c1b99392c

 Editar

 Eliminar


Configuración de API

API URL:

https://uoubqhzq31.execute-api.us-east-1.amazonaws.com/prod

API Key:

beBf21atY9AizmEkwlz2VT006oLRGm3CgpbLRu

 Guardar y Conectar

Crear / Editar Nota

Título:

Título de la nota

Contenido:

Escribe el contenido aquí...

Tags (separados por comas):

aws, lambda, cloud



Crear Nota

Nota actualizada correctamente

Mis Notas

 Recargar

Creacion Nota(Ahora editada)

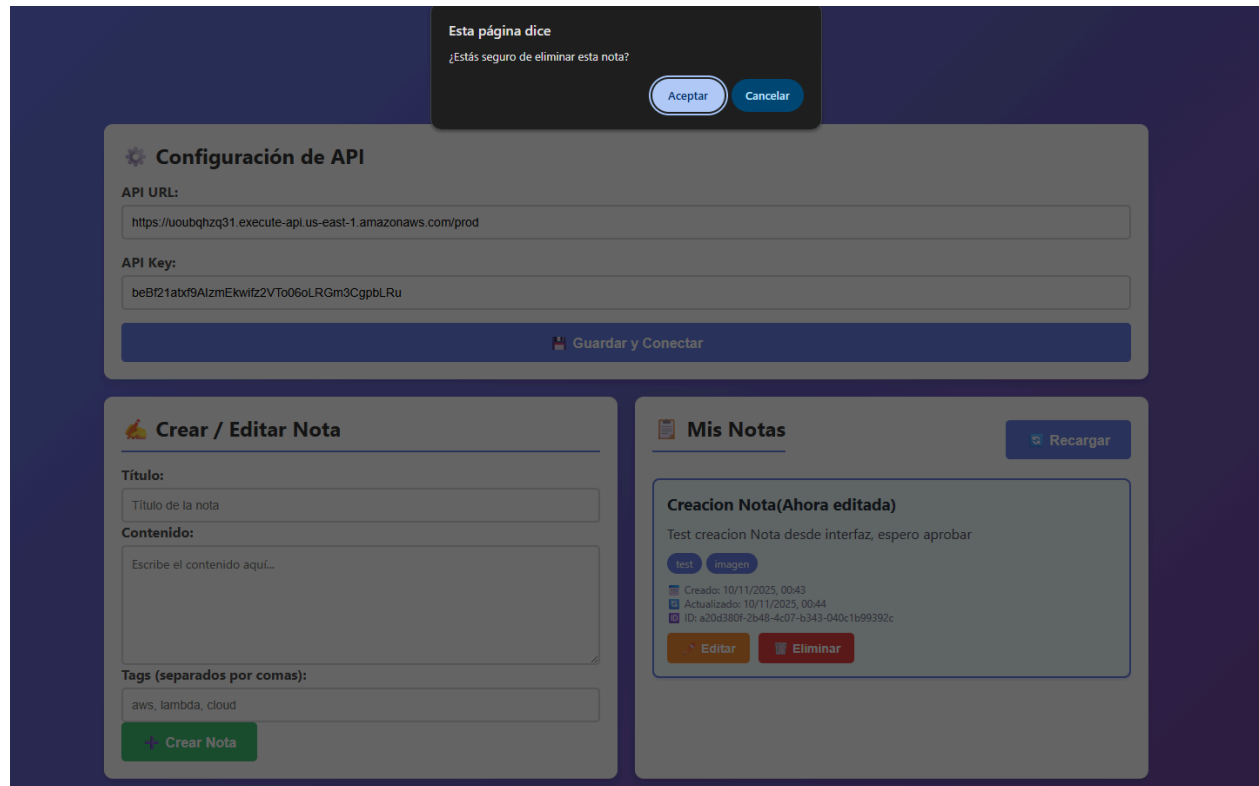
Test creacion Nota desde interfaz, espero aprobar

test imagen

 Creado: 10/11/2025, 00:43
 Actualizado: 10/11/2025, 00:44
 ID: a20d380f-2b48-4c07-b343-040c1b99392c

 Editar

 Eliminar



ANÁLISIS DE COSTOS(PRICING)

Un pilar fundamental en el diseño de arquitecturas en la nube es el análisis de costos. Para cumplir con el requisito de la práctica, se ha realizado una estimación de costos mensual y anual para ambas arquitecturas utilizando la Calculadora de Precios de AWS.

ESCENARIO DE ESTIMACIÓN

Para realizar una comparativa de costos justa, se definió un escenario hipotético de "aplicación pequeña pero activa". Los valores elegidos no buscan representar una carga de trabajo masiva, sino establecer una línea base realista para comparar cómo cada arquitectura maneja un volumen de tráfico moderado.

Los datos estimados fueron los siguientes:

- **1 Millón de Peticiones/mes:** Se eligió 1M como una línea base realista (ej: ~1.100 usuarios haciendo 30 operaciones/día).
- **1 GB Almacenamiento (DynamoDB):** Se asume un generoso tamaño medio de 1 KB por nota. Por tanto, 1 GB permitiría almacenar un millón de notas.
- **256 MB / 200 ms (Lambda):** Se eligieron 256 MB como un equilibrio estándar para una función Python con Boto3/Pydantic, que coincide con la plantilla (04-lambda-option-b.yml). 200 ms es una estimación conservadora para una llamada a la API que incluye una operación de DynamoDB

La calculadora de precios a veces omite la capa gratuita de AWS, que es sustancial. La capa gratuita perpetua de Lambda (1M de peticiones/mes) y DynamoDB (25M de peticiones/mes) y la capa gratuita de 12 meses de API Gateway (1M de peticiones/mes) significan que la Opción B tendría un costo real de \$0.00 durante el primer año.

DESGLOSE DE COSTOS

Basándonos en las configuraciones de la Calculadora de Precios de AWS las cuales se pueden comprobar en mayor detalle en el archivo **Estimación CN - Calculadora de precios de AWS.pdf**, los costos se dividen en tres categorías:

COSTOS COMUNES

Aquí encontramos los servicios comunes y aplicables a ambas opciones como:

- Amazon API Gateway: 1.000.000 de peticiones REST API al mes.
 - **Costo Estimado:** \$3.50/mes
- Amazon DynamoDB: 1 GB de almacenamiento y 1M de peticiones On-Demand.
 - **Costo Estimado:** \$1.06/mes

COSTOS ESPECÍFICOS: OPCIÓN A(FARGATE)

Esta arquitectura incurre en costos fijos (24/7) por sus componentes de cómputo y red desglosándose en:

- AWS Fargate: 1 Tarea ejecutándose 730 horas/mes con 1 GB de RAM y 0.25 vCPU.
 - **Costo Estimado:** \$10.00 / mes
- Network Load Balancer: 1 NLB ejecutándose 730 horas/mes.
 - **Costo Estimado:** \$16.49 / mes
- Amazon ECR: 1 GB de almacenamiento para la imagen Docker.
 - **Costo Estimado:** \$0.10 / mes

COSTOS ESPECÍFICOS: OPCIÓN B(LAMBDA)

Esta arquitectura incurre en costos variables, pagando solo por lo que se usa.

- AWS Lambda: 1.000.000 de ejecuciones con 256MB de RAM y una duración de 200ms.
 - **Costo Estimado:** \$0.00 / mes (El uso está cubierto por la capa gratuita perpetua).
- Amazon S3: 0.5 GB de almacenamiento para los paquetes .zip.
 - **Costo Estimado:** \$0.01 / mes.

COMPARATIVA FINAL DE COSTOS

COSTO	OPCIÓN A (FARGATE)	OPCIÓN B (LAMBDA)
Comunes (API GW + DDB)	\$4.56	\$4.56
Específicos (Cómputo + Red + Almacén)	\$26.59	\$0.01
Total Mensual	\$31.15	\$4.57
Total Anual	\$373.80	\$54.84

Esta estimación deja claro que la Opción A es un 580% más cara que la Opción B para este escenario de uso.

La razón principal de esta diferencia es el costo fijo del Network Load Balancer (\$16.49/mes) y el costo fijo del cómputo de Fargate (\$10.00/mes), que se pagan 24/7, independientemente de si la API recibe peticiones o no. La Opción B, al ser puramente serverless y basada en eventos, solo incurre en costos cuando se usa, resultando en una solución dramáticamente más rentable.

COMPARATIVA DE ARQUITECTURAS

El objetivo fundamental de esta práctica era implementar y comparar dos soluciones de *backend* distintas para una misma API REST. Habiendo desplegado y verificado con éxito tanto la Opción A como la Opción B, es posible realizar una comparativa directa sobre su complejidad, rendimiento, costo y mantenibilidad.

Ambas arquitecturas cumplieron con éxito los requisitos funcionales, conectándose a la misma base de datos DynamoDB y protegiéndose tras un API Gateway. Sin embargo, la experiencia de implementación y, sobre todo, el análisis de costos, revelan un claro ganador para este caso de uso.

CARACTERÍSTICA	OPCIÓN A (FARGATE)	OPCIÓN B (LAMBDA)
Unidad Principal	Un contenedor Docker 24/7	5 Funciones bajo demanda
Complejidad de Red	Extremadamente alta. Requirió un VpcLink, un NLB interno, y una depuración exhaustiva de Security Groups	Casi Nula. La integración entre API Gateway y Lambda es nativa y directa. No requiere gestión de VPC ni balanceadores.
Rendimiento	Rápido . Una vez desplegado, no hay latencia de "arranque en frío" .	Variable (frío/caliente). La primera petición a una función "dormida" incurre en un cold start (1-2 segundos). Las siguientes son instantáneas.
Escalabilidad	Automática, pero "lenta" (basada en minutos). Necesita lanzar nuevos contenedores para escalar.	Automática, masiva e instantánea (basada en segundos). Puede ir de 0 a miles de peticiones en paralelo
Costo Mensual Estimado	\$31.15 / mes.	\$4.57 / mes.

CONCLUSIONES FINALES

Tras analizar ambos despliegues, la Opción B (AWS Lambda) es la arquitectura ganadora y superior para este caso de uso.

La Opción A (Fargate) demostró ser una solución demasiado compleja y costosa para una API CRUD simple. El principal inconveniente no fue el cómputo en sí, sino los componentes de red necesarios para conectarlo de forma segura a API Gateway. Como revela el análisis de pricing, el costo fijo del Network Load Balancer (\$16.49/mes) y el cómputo 24/7 de Fargate (\$10.00/mes) representan el 85% aproximadamente del costo total. Esta arquitectura tendría sentido para tareas de larga duración que excedan el límite de 15 minutos de Lambda.

La Opción B (Lambda), en cambio, representa la arquitectura ideal para esta API:

1. Cumple el Requisito de Desacoplamiento: Cada función es independiente, se puede actualizar sin afectar a las demás y escala por sí sola.
2. Eficiencia de Costos: Es un 580% más barata que la Opción A. El costo se basa puramente en el uso, y como Lambda tiene una capa gratuita, el costo de cómputo es \$0.00 para nuestra estimación de 1 millón de peticiones. El costo total proviene casi exclusivamente de API Gateway y DynamoDB.
3. Simplicidad Operacional: A pesar de los desafíos iniciales de empaquetado, el proceso se simplificó y se hizo repetible una vez que se creó el script `package-lambdas.py`. Esta complejidad se limita al tiempo de lanzamiento y no al tiempo de ejecución, lo cual es preferible.

En conclusión, aunque ambas arquitecturas son funcionales, AWS Lambda elimina toda la gestión de la infraestructura de red, ofrece una escalabilidad superior y proporciona un modelo de costos óptimo, demostrando ser la solución ideal para este trabajo.

USO DE LA IA

Como se menciona en el proyecto docente de la asignatura, en este apartado se detalla el uso que se le ha dado a la Inteligencia Artificial (IA) como herramienta de apoyo para la realización de este trabajo. En este proyecto, la IA se ha utilizado como una herramienta de asistencia clave en múltiples fases:

- Explicación y solución de errores: Se empleó para obtener explicaciones detalladas sobre errores complejos encontrados durante el despliegue (especialmente con la configuración de red de Fargate) y para recibir propuestas de solución a dichos problemas.
- Desarrollo de scripts: Sirvió como asistente en el desarrollo de algunos scripts de automatización, agilizando la implementación de la lógica de despliegue.
- Diseño de arquitectura: Se consultó para la propuesta de la arquitectura del proyecto, ayudando a definir y comparar las dos opciones y los servicios necesarios para cada una.
- Explicación de conceptos: Se utilizó para clarificar conceptos técnicos específicos de los servicios de AWS que eran fundamentales para el diseño.
- Redacción del informe: La IA fue una herramienta de apoyo en la redacción de este informe, ayudando a estructurar el contenido, refinar la terminología y asegurar que el documento tuviera el toque técnico y formal requerido.
- Construcción Diagrama: Por falta de tiempo se recurrió a crear 2 diagramas con Mermaid para una mayor precisión a partir de las imágenes creadas en LucidChart.

En el prompt usado se emplearon técnicas de prompting para generar un "prompt principal" que instruyera a la IA en los objetivos deseados. Dada la considerable extensión de dicho prompt, se ha omitido su inclusión directa en este documento, pero se encuentra a disposición para su consulta.

BIBLIOGRAFÍA

- <https://docs.aws.amazon.com/toolkit-for-visual-studio/latest/user-guide/lambda-creating-project-docker-image.html>
- <https://docs.aws.amazon.com/AWSCloudFormation/latest/TemplateReference/introduction.html>
- <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-guide.html>
- <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cloudformation-supplied-parameter-types.html>
- https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html
- <https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html>
- <https://docs.aws.amazon.com/apigateway/latest/developerguide/set-up-private-integration.html>
- <https://github.com/Axelcab/P4-Aula>
- <https://gemini.google.com/app?hl=es-ES>
- <https://claude.ai/new>
- <https://chatgpt.com/>